Proof Assisted Model Checking for B*

Jens Bendisposto and Michael Leuschel

Institut für Informatik, Heinrich-Heine Universität Düsseldorf
Universitätsstr. 1, D-40225 Düsseldorf
{bendisposto,leuschel}@cs.uni-duesseldorf.de

Abstract. With the aid of the ProB Plugin, the Rodin Platform provides an integrated environment for editing, proving, animating and model checking Event-B models. This is of considerable benefit to the modeler, as it allows him to switch between the various tools to validate, debug and improve his or her models. The crucial idea of this paper is that the integrated platform also provides benefits to the tool developer, i.e., it allows easy access to information from other tools. Indeed, there has been considerable interest in combining model checking, proving and testing. In previous work we have already shown how a model checker can be used to complement the Event-B proving environment, by acting as a disprover. In this paper we show how the prover can help improve the efficiency of the animator and model checker.

Keywords: Model Checking, B-Method, Theorem Proving, Experiment, Tool Integration.

1 Introduction

There has been considerable interest in combining model checking, proving and testing (e.g., [22,23,25,11,28,4,12,15,13,14,32,29,24,5]). The Rodin platform for the formal Event-B notation provides an ideal framework for integrating these techniques. Indeed, Rodin is based on the extensible Eclipse platform and as such it is easy for provers, model checkers and other arbitrary tools to interact. In this paper we make use of this feature of Rodin to improve the ProB [18,19] model checking algorithm by using information provided by the various Rodin provers.

More concretely, in this paper we show how we can optimize the *consistency checking* of Event-B and B models, i.e., checking whether the invariants of the model hold in all reachable states. The key insight is that from the proof information we can deduce that certain events are guaranteed to preserve the correctness of specific parts of the invariant. By keeping track of which events lead to which states, we can avoid having to check a (sometimes considerable) amount of invariants.

* This research is being carried out as part of the DFG funded research project GEPAVAS and the EU funded FP7 research project 214158: DEPLOY (Industrial deployment of advanced system engineering methods for high productivity and dependability).

K. Breitman and A. Cavalcanti (Eds.): ICFEM 2009, LNCS 5885, pp. 504–520, 2009.
© Springer-Verlag Berlin Heidelberg 2009

The paper is structured as follows. In Section 2 we introduce the Event-B formal method and the Rodin platform, while in Section 3 we provide background about consistency checking and the ProB model checker, which itself already employs a combination of model checking and constraint solving techniques. In Section 4 we explain our approach to using proof information for optimizing the process of checking invariants in the ProB model checker, and present an improved model checking algorithm. Section 5 introduces a fully proven formal model of our approach. In Section 6 we evaluate our approach on a series of case studies, drawn from the Deploy project. The experiments show that there can be considerable benefit from exploiting proof information during model checking. In Section 7 we discuss how our method can be used in the context of classical B without easy access to proof information. We conclude with related work and discussions in Section 8.

2 Event-B and Rodin

Event-B is a formal method for state-based system modeling and analysis evolved from the B-method [1]. The B-method itself is derived from Z and based upon predicate logic combined with set theory and arithmetic, and provides several sophisticated data structures (sets, sequences, relations, higher-order functions) and operations on them (set union, intersection, relational composition, relational image, to name but a few).

An Event-B development consists of two types of artifacts: contexts and machines. The static properties are expressed in contexts, the dynamic properties of a system are specified in machines. A context contains definitions of carrier sets, constants as well as a set of axioms. A machine basically consists of finite sets of variables v and a finite set of events. The variables form the state of the machine, they are restricted and given a type by an invariant. The events describe transitions from one state into another state. An event has the form:

$$\text{event} \mathrel{\widehat{=}} \text{ANY } t \text{ WHERE } G(v,t) \text{ THEN } S(v,t) \text{ END}$$

It consists of a set of local variables t, a predicate $G(v,t)$, called the guard and a substitution $S(v,t)$. The guard restricts possible values for t and v. If the guard of an event is false, the event cannot occur and it is called disabled. The substitution S modifies some of the variables in v, it can use the old values of v and the local variables t. For instance, an event that chooses two natural numbers a,b and adds their product ab to the state variable $x \in v$ could be written as

$$\text{evt1} \mathrel{\widehat{=}} \text{ANY } a,b \text{ WHERE } a \in \mathbb{N} \wedge b \in \mathbb{N} \text{ THEN } x := x + ab \text{ END}$$

The Rodin tool [2] was developed within the EU funded project RODIN [26] and is an open platform for Event-B. The Rodin core puts emphasis on mathematical proof of models, while other plug-ins allow, for instance, UML-like editing, animation or model checking. The platform interactively checks a model, generates

and discharges proof obligations for Event-B. These proof obligations deal with different aspects of the correctness of a model. In this paper we only deal with proofs that are related to invariant preservation, i.e., if the invariant holds in a state and we observe an event, the invariant still holds in the successor state:

$$I(v) \wedge G(v,t) \wedge S_{BA}(v,t,v') \Longrightarrow I(v')$$

By $S_{BA}(v,t,v')$ we mean the substitution S expressed as a Before-After predicate. The primed variables refer to the state after the event happened, the unprimed variables to the state before the event happened. In our small example, $S_{BA}(v,t,v')$ is the predicate $x' = x + ab$. If we want to express, that x is a positive integer, i.e. $x \in \mathbb{N}_1$, we need to prove:

$$x \in \mathbb{N}_1 \wedge a \in \mathbb{N} \wedge b \in \mathbb{N} \wedge x' = x + ab \Longrightarrow x' \in \mathbb{N}_1$$

This implication is obviously very easy to prove, in particular, it is possible to automatically discharge this obligation using the Rodin tool.

For each pair of invariant and event the Rodin Proof Obligation Generator, generates a proof obligation (PO) that needs to be discharged in order to prove correctness of a model as mentioned before. A reasonable number of these POs are discharged fully automatically by the tool. If an obligation is discharged, we know that if we observe an event and the invariant was valid before, then it will be valid afterwards. Before generating proof obligations, Rodin statically checks the model. Because this also includes type checking, the platform can eliminate a number of proof obligations that deal with typing only. For instance the invariant $x \in \mathbb{Z}$ does not give rise to any proof obligation, its correctness is guaranteed by the type checker.

The propagation and exploitation of this kind of proof information to help the model checker is the key concept of the combination of proving and model checking presented in this paper.

3 Consistency Checking and ProB

ProB [18,19] is an animator for B and Event-B built in Prolog using constraint-solving technology. It incorporates optimizations such as symmetry reduction (see, e.g., [30]) and has been successfully applied to several industrial case studies such as a cruise control system [18], parts of the Nokia Mobile Internet Technical Architecture (MITA) and the most recent one: the application of ProB to verify the properties of the San Juan Metro System deployment [20].

One core application of ProB is the *consistency checking* of a B model, i.e., checking whether the invariant of a B machine is satisfied in all initial states and whether the invariant is preserved by the operations of the machine. ProB achieves this by computing the *state space* of a B model, by

- computing all possible initializations of a model and
- by computing for every state all possible ways to enable events and computing the effects of these events (i.e., computing all possible successor states).

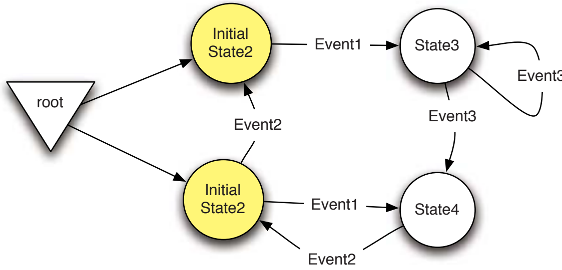

Fig. 1. A simple state space with four states

Graphically, the state space of a B model looks like in Figure 1. Note that the initial states are represented as successor states of a special root node.

ProB then checks the invariant for every state in the state space. (Note that ProB can also check assertions, deadlock absence and full LTL properties [21].)

Another interesting aspect is that ProB uses a mixture of depth-first and breadth-first evaluation of the state space, which can lead to considerable performance improvements in practice [17].

4 Proof-Supported Consistency Checking

The status of a proof obligation carries valuable information for other tools, such as a model checker. As described, ProB does an exhaustive search, i.e. it traverses the state space and verifies that the invariant is preserved in each state. This section describes how we incorporate proof information from Rodin into the ProB core.

Assuming we have a model, that contains the invariant $[I_1, I_2, I_3]$[1] and we follow an event *evt* to a new state. If we would, for instance, know that *evt* preserves I_1 and I_3, there would be no need to check these invariants. This kind of knowledge, which is precisely what we get from a prover, can potentially reduce the cost of invariant verification during the model checking.

The ProB plug-in translates a Rodin development, consisting of the model itself, its abstractions and all necessary contexts into a representation used by ProB. We evolved this translation process to also incorporate proof information, i.e., our representation contains a list of tuples (E_i, I_j) of all discharged POs, that is event E_i preserves invariant I_j.

Using all this information, we determine an individual invariant for each event that is defined in the machine. Because we only remove proven conjuncts, this specialized invariant is a subset of the model's invariant. When encountering a new state, we can evaluate the specialized invariant rather than the machine's full invariant.

[1] Sometimes it is handier to use a list of predicates rather than a single predicate, we use both notations equivalently. If we write $[P_1, P_2, \ldots, P_n]$, we mean the predicate $P_1 \wedge P2 \wedge \ldots \wedge P_n$.

As an example we can use the Event-B model shown in Figure 2. The full state space of this model and the proof status delivered by the automatic provers of the Rodin tool are shown in Figure 3.

VARIABLES
f, x
INVARIANTS
$inv1: \ f \in \mathbb{N} \nrightarrow \mathbb{N}$
$inv2: \ x > 3$
EVENTS
Initialisation
$f := \{1 \mapsto 100\} || x := 10$
Event a $\widehat{=}$
$f := \{1 \mapsto 100\} || x := f(1)$
Event b $\widehat{=}$
$f := f \cup \{1 \mapsto 100\} || x := 100$

Fig. 2. Example for intersection of invariants

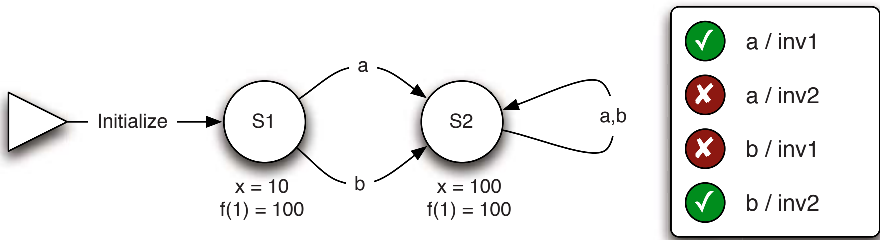

Fig. 3. State space of the model in figure 2

The proof status at the right shows, that Rodin is able to discharge the proof obligations $a/inv1$ and $b/inv2$ but not $a/inv2$ and $b/inv1$. This means, if a occurs, we can be sure that $f \in \mathbb{N} \nrightarrow \mathbb{N}$ holds in the successor state if it holds in the predecessor state. Analogously, we know, that if b occurs, we are sure, that $x > 3$ holds in the successor state if it holds in the predecessor state.

Consider a situation, where we already verified that all invariants hold for $S1$ and we are about to check $S2$ is consistent. We discovered two incoming transitions corresponding to the events a and b. From a, we can deduct that $f \in \mathbb{N} \nrightarrow \mathbb{N}$ holds. From b, we know that $x > 3$ holds. To verify $S2$, we need to check the intersection of unproven invariants, i.e., $\{f \in \mathbb{N} \nrightarrow \mathbb{N}\} \cap \{x > 3\} = \varnothing$, thus we already know that all invariants hold for $S2$.

This is of course only a tiny example but it demonstrates, that using proof information we are able to reduce the number of invariants for each event significantly, and sometimes by combining proof information from different events,

we are able to get rid of the whole invariant. We actually have evidence that this is not only a theoretical possibility, but happens in real world specifications (see Section 6).

Algorithm 4.1 [*Proof-Supported Consistency Checking*]

```
Input: An Event-B model with invariant I = inv_1 ∧ ... ∧ inv_n
Queue := {root} ; Visited := {}; Graph := {}
for all events evt do Unproven(evt) := {inv_i | inv_i not proven for evt}; end do
while Queue is not empty do
  if random(1) < α then
    state := pop_from_front(Queue); /* depth-first */
  else
    state := pop_from_end(Queue); /* breadth-first */
  end if
  if ∃inv_i ∈ Inv(state) s.t. inv_i is false then
    return counter-example trace in Graph
              from root to state
  else
    for all succ,evt such that state →_evt succ do
      Graph := Graph ∪ {state →_evt succ}
        if succ ∉ Visited then
            push_to_front(succ, Queue);
            Visited := Visited ∪ {succ}
            Inv(succ) := Unproven(evt)
        else
            Inv(succ) := Inv(succ) ∩ Unproven(evt)
        end if
      end if
    end for
od
return ok
```

Algorithm 4.1 describes ProB's consistency checking algorithm, we will justify it formally in section 5. The algorithm employs a standard queue data structure to store the unexplored nodes. The key operations are:

- Computing the successor states, i.e., "$state \rightarrow_{evt} succ$".
- Verification of the invariant "$\exists inv_i \in Inv(state)$ *s.t.* inv_i *is false*"
- Determining whether "$succ \notin$ Visited"

The algorithm terminates when there are no further queued states to explore or when an error state is discovered. The underlined parts highlight the important differences with the algorithm in [19].

In contrast to the algorithm, the actual implementation does the calculation of the intersection ($Inv(succ) := Inv(succ) \cap Unproven(op)$) in a lazy manner, i.e., for each $state \notin Visited$, we store the event names as a list. As soon as we evaluate the invariant of a state, we calculate and evaluate the intersection on the fly. The reason is, that storing the invariant's predicate for each state is typically more expensive than storing the event names.

5 Verification

To show, that our approach is indeed correct, we developed a formal model of an abstraction of algorithm 4.1. We omitted few technical details, such as the way the state space is traversed by the actual implementation and also we omitted the fact, that our implementation always uses all available information. Instead, we have proven correctness for any traversal and any subset of the available information. Our model was developed using Event-B and fully proven in Rodin. The model is available as a Rodin 1.0 archive from `http://deploy-eprints.ecs.soton.ac.uk/152/`. In this paper we present only some parts of the model and some lemmas, without their proofs. All the proofs can be found in the file, we thus refer the reader to the Rodin model.

We used three carrier sets *STATES*, *INVARIANTS* and *EVENTS*. We assume, that these sets are finite. For invariants and events this is true by definition in Event-B, but the state space can in general be unbounded. However, the assumption of only dealing with finite state spaces is reasonable in the context of our particular model, because we can interpret the *STATES* set as the subset of all states that can be traversed by the model checker within some finite number of steps.[2] The following definitions are used to prove some properties of Event-B:

$$
\begin{aligned}
&\mathit{truth} \subseteq \mathit{STATES} \times \mathit{INVARIANTS}\\
&\mathit{trans} \subseteq \mathit{STATES} \times \mathit{STATES}\\
&\mathit{preserve} = \{s \mid \{s\} \times \mathit{INVARIANTS} \subseteq \mathit{truth}\}\\
&\mathit{violate} = \mathit{STATES} \setminus \mathit{preserve}\\
&\mathit{label} \subseteq \mathit{trans} \times \mathit{EVENTS}\\
&\mathit{discharged} \subseteq \mathit{EVENTS} \times \mathit{INVARIANTS}
\end{aligned}
$$

The model also contains a set *truth*: pair of a state s and an invariant i is in *truth* if and only if i holds in s. The set *preserve* is defined as the set of states where each invariant holds, the relations *trans* and *label* describe, how two states are related, i.e. a triple $(s \mapsto t) \mapsto e$ is in *label* (and therefore $s \mapsto t \in \mathit{trans}$) if and only if t can be reached from s by executing e. The observation that is the foundation of all theorems we proved and is the following assumption:

$$
\begin{gathered}
\forall i, t \cdot (\exists s, e \cdot s \in \mathit{preserve} \wedge (s \mapsto t) \in \mathit{trans} \wedge \\
(s \mapsto t) \mapsto e \in \mathit{label} \wedge (e \mapsto i) \in \mathit{discharged}) \\
\Rightarrow (t \mapsto i) \in \mathit{truth}
\end{gathered}
$$

The assumption is, that if we reach a state t from a state s where all invariants hold by executing an event e and we know, that the invariant i is preserved by e, we an be sure, that i holds in t. This statement is what we prove by discharging an invariant proof obligation in Event-B, thus it is reasonable to assume that it holds.

[2] Alternatively, we can remove this assumption from our Rodin models. This only means that we are not be able to prove termination of our algorithm; all other invariants and proofs remain unchanged.

We are now able to prove a lemma, that will capture the essence of our proposal; it is enough to find for each invariant i one event that preserves this invariant leading from a consistent state into a state t to prove, that all invariants hold in t.

Lemma 1. $\forall t \cdot t \in STATES \wedge (\forall i \cdot i \in INVARIANTS \wedge (\exists s, e \cdot s \in preserve \wedge e \in EVENTS \wedge (s \mapsto t) \in trans \wedge (s \mapsto t) \mapsto e \in label \wedge e \mapsto i \in discharged)) \Rightarrow t \in preserve$

Proof. All proofs have been done using Rodin and can be found in the model archive. □

We used five refinement steps to prove correctness of our algorithm. We will describe the first three steps, the last two steps are introduced to prove termination of new events. The first refinement step *mc0* contains two events *check_state_ok* and *check_state_broken*. The events take a yet unprocessed state and move it either into a set containing consistent or inconsistent states. Algorithm 5.1 shows the *check_state_ok* event, *check_state_broken* is defined analogously, except that it has the guard $s \notin preserve$ and it puts the state into the set *inv_broken*.

Algorithm 5.1 [Event *check_state_ok* from *mc0*]

event *check_state_ok*
  any s
  where
    $s \in open$
    $s \in preserve$
  then
    $inv_ok := inv_ok \cup \{s\}$
    $open := open \setminus \{s\}$
end

At this very abstract level this machine specifies that our algorithm separates the states into two sets. If they belong to *preserve*, the states are moved into the set *inv_ok*. Otherwise, they are moved into *inv_broken*. Lemma 2 guarantees, that our model always generate correct results.

Lemma 2. *mc0 satisfies the invariants*

1. $inv_ok \cup inv_broken = STATES \setminus open$
2. $open = \varnothing \Rightarrow inv_ok = preserve \wedge inv_broken = violate$

The next refinement strengthens the guard and removes the explicit knowledge of the sets *preserve* and *violate*, the resulting proof obligation leads to lemma 3.

Lemma 3. *For all* $s \in open$

$$\{s\} \times INVARIANTS \setminus discharged[label[inv_ok \lhd trans \rhd \{s\}]]) \subseteq truth$$
$$\Leftrightarrow s \in preserve$$

The third refinement introduces the algorithm. We introduce a new relation *invs_to_verify* in this refinement. The relation keeps track of those invariants, that need to be checked, in the initialization, we set $invs_to_verify := STATES \times INVARIANTS$.

The algorithm has three different phases. It first selects a state that has not been processed yet then it checks if the invariant holds and moves the state into either *inv_ok* or *inv_broken*. Finally, it uses the information about discharged proofs to remove some elements from *invs_to_verify* as shown in algorithm 5.2.

Algorithm 5.2 [Event *mark_successor* from *mc2*]

event *mark_successor*
  any p s e
  where
    $p \in inv_ok$
    $s \in trans[\{p\}]$
    $(p \mapsto s) \mapsto e \in label$
    $(p \mapsto s) \mapsto e \notin marked$
    $ctrl = mark$
  then
    $invs_to_verify := invs_to_verify \Leftarrow (\{s\} \times (invs_to_verify[\{s\}] \cap unproven[\{e\}]))$
    $marked := marked \cup \{(p \mapsto s) \mapsto e\}$
end

We take some state s and event e, where we know that s is reachable via e from a state p, where all invariants hold. Then we remove all invariants but those, that are not proven to be preserved by e. This corresponds to the calculation of the intersection in algorithm 4.1.

The main differences between the formal model and our implementation are, that the model does not explicitly describe how the states are chosen and the algorithm uses all available proof information while the formal model can use any subset. In addition, the model does not stop if it detects an invariant violation. We did not specify these details because it causes technical difficulties (e.g., we need the transitive closure of the *trans* relation) but does not seem to provide enough extra benefit.

Correctness of algorithm 4.1 is established by the fact that the outgoing edges of a *state* are added to the *Graph* only *after* the invariants have been checked for *state*. Hence, the removal of a preserved invariant only occurs *after* it has been established that the invariant is true before applying the event. This corresponds to the guard $p \in inv_ok$. However, the proven proof obligations for an event only guarantee *preservation* of a particular invariant, not that this invariant is established by the event. Hence, if the invariant is false before applying the event, it could be false after the event, *even* if the corresponding proof obligation is proven and true. If one is not careful, one could easily set up cyclic dependencies and our algorithm would incorrectly infer that an incorrect model is correct.

6 Experimental Results

To verify that the combination of proving and model checking results in a considerable reduction of model checking effort, we prepared an experiment consisting of specifications we got from academia and industry. In addition we prepared a constructed example as one case, where the prover has a very high impact on the performance of the model checker. The rest of this section describes how we carried out the measurement. We will also briefly introduce the models and discuss the result for each of them. The experiment contains models where we expected to have a reasonable reduction and models where we expected to have only a minor or no impact.

6.1 Measurement

The latest development versions of ProB can do consistency checking of a refinement chain. Previous versions of ProB checked a specific refinement level only and removed all gluing invariants. We carried out both, single refinement level and multiple refinement level checks. The results have been gathered using a Mac Book Pro, 2.4 GHz Intel Core 2 Duo Computer with 4 GB RAM running Mac OS X 10.5. For the single level animation, we collected 40 samples for each model and calculated the average and standard deviation of the times measured in milliseconds. For the multi level animation, we collected 5 samples for each model. The result of the experiment is shown in tables 1, 2 and 3. The absolute values of tables 1 and 2 are very difficult to compare, because we used different versions of ProB.

Except for the case of the Siemens specification, we removed all interactive proofs from the models and used only those proof information, that Rodin was able to automatically generate using default settings. In the case of the Siemens model, we used both, a version with automatic proofs only and a development version with few additional interactive proofs; the development version was not fully proven.

6.2 Mondex

The mechanical verification of the Mondex Electronic Purse was proposed for the repository of the verification grand challenge in 2006. We use an Event-B model developed at the University of Southampton [8]. We have chosen two refinements from the model, m2 and m3. The refinement m2 is a rather big development step while the second refinement m3 was used to prove convergence of some events introduced in m2, in particular, m3 only contains gluing invariants.

In case of single refinement level checking, it is obvious that it is not possible to further simplify the invariant of m3 but we noticed, that we do not even lose performance caused by the additional specialization of the invariants. This is important because it is evidence, that our implementation's performance is in the order of the standard deviation in our measurement. For the case of m2, where we have machine invariants, we measured a reduction of about 12%.

In case of multiple refinement level checking, we have the only case, where we lost a bit of performance for m2. However, the absolute value is in the order of the standard deviation. For m3 we also did not get significant improvements of performance, most likely because the gluing invariant is very simple, actually it only contains simple equalities.

6.3 Siemens Deploy Mini Pilot

The Siemens Mini Pilot was developed within the Deploy Project. It is a specification of a fault-tolerant automatic train protection system, that ensures that only one train is allowed on a part of a track at a time. The Siemens model shows a very good reduction, as the invariants are rather complex. This model does contain a single machine, thus multi level refinement checking does not affect the speedup.

6.4 Scheduler

This model is an Event-B translation of the scheduler from [16]. The model describes a typical scheduler that allows a number of processes to enter a critical section. The experiment has shown, that the improvement using proof information is rather small, which was no surprise. The model has a state space that grows exponential when increasing the number of processes. It is rather cheap to check the invariant

$$ready \cap waiting = \varnothing \wedge active \cap (ready \cup waiting) = \varnothing \wedge active = \varnothing \Rightarrow ready = \varnothing$$

because the number of processes is small compared to the number of states. But nevertheless, we save a small amount of time in each state and these savings can sum up to a reasonable speedup. The scheduler also contains a single level of refinement.

6.5 Earley Parser

The model of the Earley parsing algorithm was developed and proven by Abrial. Like in the mondex example, we used two refinement steps that have different purposes. The second refinement step m2 contains a lot of invariants, while the m3 contains only very few of them. This is reflected in the savings we gained from using the proof information in the case of single refinement level checking. While m3 showed practically no improvement, in the m2 model the savings sum up to a reasonable amount of time. In the case of multiple refinement level checking the result are very different, while m2 is not affected, the m3 model benefits a lot. The reason is, that it contains several automatically proven gluing invariants.

6.6 SAP Deploy Mini Pilot

Like the Siemens model this is a Deploy pilot project. It is a model of system that coordinates transactions between seller and buyer agents. In the case of

single refinement level case, we gain a very good speedup from using proof information, i.e., model checking takes less than half of the time. Like in the Siemens example, the model contains rather complicated invariants. In case of the multi refinement level checking the speedup is still good, but not as impressive as in single refinement level checking.

6.7 SSF Deploy Mini Pilot

The Space Systems Finland example is a model of a subsystem used for the ESA BepiColombo mission. The BepiColombo spacecraft will start in 2013 on its journey to Mercury. The model is a specification of parts of the BepiColombo On-Board software, that contains a core software and two subsystems used for tele command and telemetry of the scientific experiments, the Solar Intensity X-ray and particle Spectrometer (SIXS) and the Mercury Imaging X-ray Spectrometer (MIXS). The time for model checking could be reduced by 7% for a single refinement level and by 16% for multiple refinement checking.

6.8 Cooperative Crosslayer Congestion Control CXCC

CXCC [27] is a cross-layer approach to prevent congestion in wireless networks. The key concept is that, for each end-to-end connection, an intermediate node may only forward a packet towards the destination after its successor along the route has forwarded the previous one. The information that the successor node has successfully retrieved a package is gained by active listening. The model is described in [6]. The invariants used in the model are rather complex and thus we get a good improvement by using the proof information in both cases.

6.9 Constructed Example

The constructed example is mainly to show a case, where we get a huge saving from using the proofs. It basically contains an event, that increments a number x and an invariant $\forall a, b, c \cdot a \in \mathbb{N} \wedge b \in \mathbb{N} \wedge c \in \mathbb{N} \Rightarrow (a = a \wedge b = b \wedge c = c \wedge x = x)$. Because the invariant contains the variable modified by the event, we cannot simply remove it. But Rodin can automatically prove that the event preserves the invariant, thus our tool is able to remove the whole invariant. Without proof information, ProB needs to enumerate all possible values for a,b and c which results in an expensive calculation.

7 Proof-Assisted Consistency Checking for Classical-B

In the setting of Event-B and the Rodin platform, ProB can rely on the other tools for providing type inference and as we have seen the proof information.

In the context of classical B, we are working on a tighter integration with Atelier B [31]. However, at the moment ProB does not have access to the proof information of classical B models.

ProB does perform some additional analyses of the model and annotates the AST (Abstract Syntax Tree) with additional information. For instance for each

Table 1. Experimental results (single refinement level check)

	w/o proof information [ms]	using proof information [ms]	Speedup-Factor
Mondex m3	1454 ± 5	1453 ± 5	1.00
Earley Parser m3	2803 ± 8	2776 ± 7	1.01
Earley Parser m2	140310 ± 93	131045 ± 86	1.07
SSF	31242 ± 64	29304 ± 44	1.07
Scheduler	9039 ± 15	8341 ± 14	1.08
Mondex m2	1863 ± 7	1665 ± 6	1.12
Siemens (auto proof)	54153 ± 50	25243 ± 22	2.15
Siemens	56541 ± 57	26230 ± 28	2.16
SAP	18126 ± 18	8280 ± 14	2.19
CXCC	18198 ± 21	6874 ± 12	2.65
Constructed Example	18396 ± 26	923 ± 8	19.93

Table 2. Experimental results (multiple refinement level check)

	w/o proof information [ms]	using proof information [ms]	Speedup-Factor
Mondex m2	1747 ± 21	1767 ± 38	0.99
Mondex m3	1910 ± 20	1893 ± 6	1.01
Earley Parser m2	309810 ± 938	292093 ± 1076	1.06
Scheduler	9387 ± 124	8167 ± 45	1.15
SSF	35447 ± 285	30590 ± 110	1.16
SAP	50783 ± 232	34927 ± 114	1.45
Earley Parser m3	7713 ± 40	5047 ± 15	1.53
Siemens (auto proof)	51560 ± 254	24127 ± 93	2.14
Siemens	51533 ± 297	23677 ± 117	2.18
CXCC	18470 ± 151	6700 ± 36	2.76
Constructed Example	18963 ± 31	967 ± 6	19.61

event we calculate a set of variables that are possibly modified. For instance if we analyze the operation[3]

$$Operation1 = BEGIN\ x := z \mid\mid y := y \wedge \{x \mapsto z\}\ END$$

the analysis will discover that the set of variables that could potentially influence the truth value of the invariant is $\{x, y\}$.

This analysis was originally used to verify the correct usage of `SEES` in the classical B-Method. The `SEES` construct was used in the predecessor of Event-B, so-called classical B, to structure different models. In classical B a machine can see another machine, i.e., it is allowed to call operations that do not modify the state of the other machine. To support this behavior, it was necessary to know if an operation has effect on state variables, that is the set of modified

[3] Operations are the equivalent of events in classical B.

Table 3. Number of invariants evaluated (single refinement level check)

	w/o Proof [#]	**w Proof [#]**	**Savings [%]**
Earley Parser m2	–	–	-
Mondex m3	440	440	0
Earley Parser m3	540	271	50
Constructed Example	42	22	50
SAP	48672	16392	66
Scheduler	20924	5231	75
Mondex m2	6600	1560	76
SSF	24985	5009	80
CXCC	88480	15368	83
Siemens	280000	10000	96
Siemens (auto proof)	280000	10000	96

variables is the empty set. It turned out, that the information is more valuable than originally thought, as it is equivalent to some proof obligation:

If u and v are disjoint sets of state variables, and the substitution of an operation is $S_{BA}(v, t, v')$ we know that $u = u'$ and thus a simplified proof obligation for the preservation of an invariant $I(u)$ over the variables u is

$$I(u) \wedge G(u \cup v, t) \wedge S_{BA}(v, t, v') \Rightarrow I(u)$$

which is obviously true. These kind of proof obligations are not generated by any of the proving environments for B we are aware of. In particular Rodin does not generate them. For a proving environment, this is a good idea as they do not contain valuable information for the user and they can be filtered out by simple syntax analysis. But for the model checker these proofs are very valuable; in most cases they allow us to reduce the number of invariants we need to check. As this type of proof information can be created from the syntax, we can use them even if we do not get proof information from Rodin, i.e., when working on classical B machines. As such, we were able to use Algorithm 4.1 also for classical B models and also obtain improvements of the model checking performance (although less impressive than for Event-B).

8 Conclusion and Future Work

First of all, we never found a model where using proof information significantly reduced the performance, i.e., the additional costs for calculating individual invariants for each state are rather low. Using proof information is the new default setting in ProB.

We got a number of models, in particular those coming from industry, where using the proof information has a high impact on the model checking time. In other cases, we gained only a bit or no improvement. This typically happens if the invariant is rather cheap to evaluate compared to the costs of calculating the

guards of the events. We used an out-of-the-box version of Rodin[4] to produce our experimental results. Obviously, it is possible to further improve them by adding manual proof effort. In particular, it gives the user a chance to influence the speed of the model checker by proving invariant preservation for those parts that are difficult to evaluate, i.e., those predicates that need some kind of enumeration.

Related Work. A similar kind of integration of theorem proving into a model checker was previously described in [25]. In their work Pnueli and Shahar introduced a system to verfify CTL and LTL properties. This system works as a layer on top of CMU SMV and was sucessfully applied to fragments of the Futurebus+ system [10]. SAL is a framework and tool to combine different symbolic analysis [28], and can also be viewed as an integration of theorem proving and model checking. Mocha [3] is another work where a model checker is complemented by proof, mostly for assume-guarantee reasoning. Some more works using theorem proving and model checking together are [11,4,12,15].

In the context of B, the idea of using a model checker to assist a prover has already been exploited in practice. For example, in previous work [7] we have already shown how a model checker can be used to complement the proving environment, by acting as a disprover. In [7] it was also shown that sometimes the model checker can be used as a prover, namely when the underlying sets of the proof obligation are finite. This is for example the case for the vehicle function mentioned in [18]. Another example is the Hamming encoder in [9], where Dominique Cansell has used ProB to prove certain theorems which are difficult to prove with a classical prover (due to the large number of cases).

Future Work. We have done but a first step towards exploiting the full potential for integrating proving and model checking. For instance, we may feed the theorem prover with proof obligations generated by the model checker in order to speed up the model checking. A reasonable amount of time is spent evaluating the guards. If the model checker can use the theorem prover to prove that an event e is guaranteed to be disabled after an event f occurs, we can reduce the effort of checking guards. We may need to develop heuristics to find out when the model checker should try to get help from the provers.

Also we might feed information from the model checker back into the proving environment. If the state space is finite and we traverse all states, we can use this as a proof for invariant preservation. ProB restricts all sets to finite sets [19] to overcome the undecidability of B, so this needs to be handled with care. We need to ensure, that we do not miss states because ProB restricted some sets. Also we need to ensure that all states are reachable by the model checker, thus we may need some additional analysis of the model.

We also think of integrating a prover for classical B, to exploit proof information. The integration is most likely not as seamless as in Rodin and the costs of getting proof information is higher.

[4] For legal reasons, it is necessary to install the provers separately.

Although the cost of calculating the intersections of the invariants for each state is too low to measure it, the stored invariants take some memory. It might be possible to find a more efficient way to represent the intersections of invariants.

References

1. Abrial, J.-R.: The B-Book. Cambridge University Press, Cambridge (1996)
2. Abrial, J.-R., Butler, M., Hallerstede, S.: An open extensible tool environment for Event-B. In: Liu, Z., He, J. (eds.) ICFEM 2006. LNCS, vol. 4260, pp. 588–605. Springer, Heidelberg (2006)
3. Alur, R., Henzinger, T.A., Mang, F.Y.C., Qadeer, S., Rajamani, S.K., Tasiran, S.: Mocha: Modularity in model checking. In: Hu, A.J., Vardi, M.Y. (eds.) CAV 1998. LNCS, vol. 1427, pp. 521–525. Springer, Heidelberg (1998)
4. Arkoudas, K., Khurshid, S., Marinov, D., Rinard, M.C.: Integrating model checking and theorem proving for relational reasoning. In: Berghammer, R., Möller, B., Struth, G. (eds.) RelMiCS 2003. LNCS, vol. 3051, pp. 21–33. Springer, Heidelberg (2004)
5. Arons, T., Pnueli, A., Zuck, L.: Parameterized verification by probabilistic abstraction. In: Gordon, A.D. (ed.) FOSSACS 2003. LNCS, vol. 2620, pp. 87–102. Springer, Heidelberg (2003)
6. Bendisposto, J., Jastram, M., Leuschel, M., Lochert, C., Scheuermann, B., Weigelt, I.: Validating Wireless Congestion Control and Realiability Protocols using ProB and Rodin. FMWS 2008: Workshop on Formal Methods for Wireless Systems (August 2008)
7. Bendisposto, J., Leuschel, M., Ligot, O., Samia, M.: La validation de modèles event-b avec le plug-in prob pour rodin. Technique et Science Informatiques 27(8), 1065–1084 (2008)
8. Butler, M., Yadav, D.: An incremental development of the Mondex system in Event-B. Formal Aspects of Computing 20(1), 61–77 (2008)
9. Cansell, D., Hallerstede, S., Oliver, I.: UML-B specification and hardware implementation of a hamming coder/decoder. In: Mermet, J. (ed.) UML-B Specification for Proven Embedded Systems Design, ch. 16, November 2004. Kluwer Academic Publishers, Dordrecht (2004)
10. Clarke, E., Grumberg, O., Hiraishi, H., Jha, S.: Verification of the futurebus+ cache coherence protocol. Formal Methods in System Design (January 1995)
11. Dams, D., Hutter, D., Sidorova, N.: Using the inka prover to automate safety proofs in abstract interpretation - a case study. In: Bellegarde, F., Kouchnarenko, O. (eds.) Workshop on Modelling and Verification, C.I.S., Besançon, France (1999); Alternative title: Combining Theorem Proving and Model Checking - A Case Study
12. Dybjer, P., Haiyan, Q., Takeyama, M.: Verifying haskell programs by combining testing, model checking and interactive theorem proving. Information & Software Technology 46(15), 1011–1025 (2004)
13. Freitas, L.: Model Checking Circus. PhD thesis, University of York (2005)
14. Freitas, L., Woodcock, J., Cavalcanti, A.: State-rich model checking. Innovations Syst. Softw. Eng. 2(1), 49–64 (2006)
15. Gunter, E.L., Peled, D.: Model checking, testing and verification working together. Formal Asp. Comput. 17(2), 201–221 (2005)
16. Legeard, B., Peureux, F., Utting, M.: Automated boundary testing from Z and B. In: Eriksson, L.-H., Lindsay, P.A. (eds.) FME 2002. LNCS, vol. 2391, pp. 21–40. Springer, Heidelberg (2002)

17. Leuschel, M.: The high road to formal validation. In: Börger, E., Butler, M., Bowen, J.P., Boca, P. (eds.) ABZ 2008. LNCS, vol. 5238, pp. 4–23. Springer, Heidelberg (2008)
18. Leuschel, M., Butler, M.: ProB: A model checker for B. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 855–874. Springer, Heidelberg (2003)
19. Leuschel, M., Butler, M.J.: ProB: an automated analysis toolset for the B method. STTT 10(2), 185–203 (2008)
20. Leuschel, M., Falampin, J., Fritz, F., Plagge, D.: Automated property verification for large scale b models. In: Cavalcanti, A., Dams, D. (eds.) Proceedings FM 2009. LNCS, vol. 5850, pp. 708–723. Springer, Heidelberg (2009)
21. Leuschel, M., Plagge, D.: Seven at a stroke: LTL model checking for high-level specifications in B, Z, CSP, and more. In: Ameur, Y.A., Boniol, F., Wiels, V. (eds.) Proceedings Isola 2007, Cépaduès edn. Revue des Nouvelles Technologies de l'Information, vol. RNTI-SM-1, pp. 73–84 (2007)
22. Müller, O., Nipkow, T.: Combining model checking and deduction for i/o-automata. In: Brinksma, E., Steffen, B., Cleaveland, W.R., Larsen, K.G., Margaria, T. (eds.) TACAS 1995. LNCS, vol. 1019, pp. 1–16. Springer, Heidelberg (1995)
23. Owre, S., Rajan, S., Rushby, J., Shankar, N., Srivas, M.: PVS: Combining specification, proof checking, and model checking. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 411–414. Springer, Heidelberg (1996)
24. Pnueli, A., Ruah, S., Zuck, L.: Automatic deductive verification with invisible invariants. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 82–97. Springer, Heidelberg (2001)
25. Pnueli, A., Shahar, E.: A platform for combining deductive with algorithmic verification. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 184–195. Springer, Heidelberg (1996)
26. Romanovsky, A.: Rigorous Open Development Environment for Complex Systems - RODIN. ERCIM News 65, 40–41 (2006)
27. Scheuermann, B., Lochert, C., Mauve, M.: Implicit hop-by-hop congestion control in wireless multihop networks. In: Ad Hoc Networks (2007), doi:10.1016/j.adhoc.2007.01.001
28. Shankar, N.: Combining theorem proving and model checking through symbolic analysis. In: Palamidessi, C. (ed.) CONCUR 2000. LNCS, vol. 1877, pp. 1–16. Springer, Heidelberg (2000)
29. Sipma, H., Uribe, T., Manna, Z.: Deductive model checking. Formal Methods in System Design 15(1), 49–74 (1999)
30. Spermann, C., Leuschel, M.: ProB gets nauty: Effective symmetry reduction for B and Z models. In: Proceedings Symposium TASE 2008, Nanjing, China, pp. 15–22. IEEE, Los Alamitos (2008)
31. Steria, F.: Aix-en-Provence. In: Atelier B, User and Reference Manuals (1996), `http://www.atelierb.societe.com`
32. Uribe, T.: Combinations of model checking and theorem proving. In: Kirchner, H., Ringeissen, C. (eds.) Frocos 2000. LNCS (LNAI), vol. 1794, pp. 151–170. Springer, Heidelberg (2000)